RSA周边——大素数是怎样生成的？

2014-07-19 宾狗 理论 数论

0x00 前言

学计算机的童鞋们一定知道RSA算法，学数学的小伙们更是不用说了（人家就是学这个好么-_-）。当然这次我们不讨论RSA算法本身（什么你还不懂RSA是怎么回事？！自己面壁思过去>_<），关于RSA算法的教材、代码满天飞，互联网上科普文更是一抓一大把，且不管它们质量如何，反正这口冷饭再炒也没什么意思，能真正攻破它才是王道……好吧，扯远了，对于我们这些普通程序员来说能用好RSA算法就行了，当然能理解它就更好了。

下面进入正题，我们都知道，RSA算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，但是想要对其乘积进行因式分解却极其困难。那么也就是说要使用RSA算法，前提是必须有两个大素数才行，那么你有没有想过大素数是怎么生成的呢？更进一步，考虑到RSA算法的高效性和安全性，怎样快速生成一个大素数呢？如果这些问题成功的引起了你的好奇心，那就接着往下看吧~

0x01 素性测试

我们先从一个更简单的问题开始讨论，即给定一个大整数，怎样判定其是不是素数？

有的同学说这还不简单，学过C语言的都知道怎么写，如果要判定 `n` 是否是素数，写一个循环从 `2` 到 `sqrt(n)` 判断其能否整除 `n` ，若都不能则 `n` 为素数。这个方法我们一般称之为**试除法**

怎么说呢，如果 `n` 比较小的话，采用试除法当然是非常高效快捷的。但是当 `n` **很大**的时候，这个算法可就行不通了，以RSA1024为例，当公钥为

```
0x890e23101a542913da8a4350672c9ef8e7b34c2687ce8cd8db3fb34244a791d60c9dc0a53172a56dcc8a66f553c0ae
51e9e2e2ce9486fa6b00a6c556bfed139001133cdfe5921c425eb8823b1bd0a4c00920d24bee2633256328502eadbfac
1420f9a5f47139de6f14d8eb7c2b7c0cec42530c0a71dadb80c7214f5cd19a3f2f
```

时，两个质因数分别为

```
0xe5a111a219c64f841669400f51a54dd4e75184004f0f4d21c6ae182cfb528652a02d6d677a72b564c505b1ed42a0c6
48dbfe14eb66b04c0d60ba3872826c32e7
```

和

```
0x98cb760764484e29245521be08e7f38edeebfca8427149524ba7f4735e1d5f3a45d585cb3722ff4c07c19165be7383
11dc346a914966f5b311416fed3b425079
```

转换为十进制分别为

```
120266557722106794704655816090025253292457737321320147427589355111878634879190264570762529320486
19706498126046597130520643092209728783224795661331197604583
```

和

```
800251142659642435182926709953165139044805415345232118535074684530627758585667389804874041343944
2356860630765545600353049345324913056448174487017235828857
```

这是一个 `155` 位数和一个 `154` 位数，都在 `2` 的 `511` 次方左右，如果看到这里你还觉得试除法可行的话，我只想问：你家是有超算么？！

既然没有超算，试除法显然是行不通的。那么怎么办呢？不用担心，你要相信，聪明的数学家们总是能想出各种神奇的办法。不过在此之前我们还是先恶补一下其中用到的一些知识(^_^)

费马小定理

直接看维基百科的解释

> **费马小定理**是数论中的一个定理：假如a是一个整数，p是一个质数，那么$a^p - a$是p的倍数，可以表示为 $a^p \equiv a \pmod{p}$ 如果a不是p的倍数，这个定理也可以写成 $a^{p-1} \equiv 1 \pmod{p}$

费马小定理的证明过程小伙伴们可以参考其他资料，这里就不再赘述

需要注意的是，费马小定理是判定一个数是否为素数的**必要条件**，并非充分条件，因为存在着一些**伪素数**满足费马小定理却不是素数，如$2^{340} \equiv 1 \pmod{341}$，但是$341 = 11 \times 31$

Fermat素性测试

刚才我们只考虑了`a=2`的情况，如果我们考虑`a=3`的情况，一个合数可能在`a=2`时通过了测试，但`a=3`时的计算结果却排除了素数的可能。于是，人们扩展了伪素数的定义，称满足$a^{n-1} \bmod n = 1$的合数`n`叫做以`a`为底的伪素数(pseudoprime to base a)。前10亿个自然数中同时以2和3为底的伪素数只有1272个，这个数目不到刚才的$\frac{1}{4}$。这告诉我们如果同时验证`a=2`和`a=3`两种情况，算法出错的概率降到了`0.000025`。容易想到，选择用来测试的`a`越多，算法越准确。通常我们的做法是，随机选择若干个小于待测数的正整数作为底数`a`进行若干次测试，只要有一次没有通过测试就可以判定这个数为合数。这就是Fermat素性测试。

人们自然会想，如果考虑了所有小于`n`的底数`a`，出错的概率是否就可以降到`0`呢？遗憾的是，居然就有这样的合数，它可以通过所有`a`(前提是`n`与`a`互素)的测试。Carmichael第一个发现这样极端的伪素数，他把它们称作Carmichael数。你一定会以为这样的数一定很大。错。第一个Carmichael数小得惊人，仅仅是一个三位数，`561`。前10亿个自然数中Carmichael数也有600个之多。Carmichael数的存在说明，我们还需要继续加强素性判断的算法。

Miller-Rabin素性测试

还是先来看看维基百科给出的解释

> 要测试N是否为素数，首先将$N-1$分解为$2^s d$。在每次测试开始时，先随机选一个介于$[1, N-1]$的整数a，之后如果对所有的$r \in [0, s-1]$，若$a^d \mod N \neq 1$且$a^{2^r d} \mod N \neq -1$，则N是合数。否则，N有$\frac{3}{4}$的概率为素数。

先不管有没有看懂，你一定注意到了最后一句话。没错，和Fermat素性测试一样，Miller-Rabin素性测试依然只能判定一个数**可能是素数**，但是这个方法却以它的快速、高效而广为使用

下面我们换个说法来解释一下Miller-Rabin素性测试的过程

首先这个待测数N一定是一个奇数（如果是偶数的话你还测它干嘛?-_-），所以$N-1$是一个偶数，而且一定可以写成$2^s d$的形式，这一步非常简单也很好理解。然后回顾Fermat素性测试，这里也类似，选取一个介于$[1, N-1]$的整数a作为底。刚才我们把$N-1$写成了$2^s d$的形式，得到了s和d，如果$a^d \mod N = 1$，或者存在一个$r \in [0, s-1]$，使得$a^{2^r d} \mod N = -1$（和$a^{2^r d} \mod N = N-1$是等价的），那么我们就称N通过了以a为底的Miller-Rabin素性测试，也就是说N有很大可能是素数。

> 证明的过程中用到了**二次探测定理**，定理很简单，如果p是一个素数，由$x^2 \equiv 1 \pmod{p}$可知$x \equiv 1 \pmod{p}$或$x \equiv p-1 \pmod{p}$
>
> 这是显然的，因为$x^2 \equiv 1 \pmod{p}$相当于p能整除x^2-1，即$(x+1)(x-1)$，由于p是素数，那么只可能是$x-1$能被p整除（此时$x \equiv 1 \pmod{p}$）或者$x+1$能被p整除（此时$x \equiv p-1 \pmod{p}$）

其实Miller-Rabin素性测试的原理就是在Fermat素性测试的基础上添加了一个二次探测的过程，有兴趣的小伙伴可以参考相关资料中的详细解释~

##Miller-Rabin素性测试的实现 了解了Miller-Rabin素性测试的基本原理就可以用程序来实现它，先来看看伪代码

```
Input: n > 3, an odd integer to be tested for primality;
Input: k, a parameter that determines the accuracy of the test
Output: composite if n is composite, otherwise probably prime
write n - 1 as 2^s·d with d odd by factoring powers of 2 from n - 1
LOOP: repeat k times:
   pick a randomly in the range [2, n - 2]
   x ← a^d mod n
   if x = 1 or x = n - 1 then do next LOOP
   for r = 1 .. s - 1
      x ← x^2 mod n
      if x = 1 then return composite
      if x = n - 1 then do next LOOP
   return composite
return probably prime
```

思路很简单，对照前面的解释很容易看懂~

本来想展示一个`Java`版的程序，结果发现`Java`中的`java.math.BigInteger`已经内置了Miller-Rabin素性测试函数

```
public boolean isProbablePrime(int certainty)
```

注意`certainty`这个参数表示调用者能容忍的不确定性，如果该函数返回`True`，那么表示这个数是素数的概率是$1-\frac{1}{2^{certainty}}$，`certainty`参数越大，该数是素数的可能性越大，相应的执行时间越长

我们来看看用`Python`怎么实现Miller-Rabin素性测试

```python
def is_probable_prime(n, trials = 5):
    assert n >= 2
    # 2是素数~
    if n == 2:
        return True
    # 先判断n是否为偶数，用n&1==0的话效率更高
    if n % 2 == 0:
        return False
    # 把n-1写成(2^s)*d的形式
    s = 0
    d = n - 1
    while True:
        quotient, remainder = divmod(d, 2)
        if remainder == 1:
            break
        s += 1
        d = quotient
    assert(2 ** s * d == n - 1)

    # 测试以a为底时，n是否为合数
    def try_composite(a):
        if pow(a, d, n) == 1: # 相当于(a^d)%n
            return False
        for i in range(s):
            if pow(a, 2 ** i * d, n) == n - 1: #相当于(a^((2^i)*d))%n
                return False
        return True # 以上条件都不满足时，n一定是合数

    # trials为测试次数，默认测试5次
    # 每次的底a是不一样的，只要有一次未通过测试，则判定为合数
    for i in range(trials):
        a = random.randrange(2, n)
        if try_composite(a):
            return False

    return True #通过所有测试，n很大可能为素数
```

代码的解释在注释中写的很详细了，这里就不再啰嗦了~

#0x02 大素数的生成 了解了Miller-Rabin素性测试的原理，下面就可以探讨如何生成大素数了。

其实生成一个大素数非常简单，最直观的方法就是随机搜索，例如要生成一个100位的大素数，我们先随机生成一个数字序列，然后用Miller-Rabin素性测试对其进行测试即可，如果不是素数的话再随机生成一个，如此循环下去~

当然我们可以采用随机搜索法（每次生成一个完全不一样的随机数），也可以采用随机递增搜索法（生成一个随机数之后，每次对其加2）

生成一个`n`位十进制大素数的步骤如下：

1. 产生一个`n`位的随机数`p`，且最高位不能为`0`
2. 若最低位为偶数，则将它加`1`，保证该数为奇数以节省时间
3. 测试该数能否被`10000`以下的素数（共`1228`个）整除，这样可以快速排除许多合数，节省时间
4. 在`2`到`p-1`这间随机生成一个数`a`，以`a`为底对`p`进行Miller-Rabin素性测试，若不通过说明`p`为合数。若通过则再选取一个`a`对`p`进行测试。选取`a`时应该选取尽可能小的素数，以提高运算速度。大概进行`5`次Miller-Rabin素性测试后，精确性就比较高了
5. 若`p`每次测试都通过，则认为`p`是素数。否则`p←p+2`，再次对`p`进行测试

下面我们来看看网上的一个开源 `Javascript` 加密函数库（新浪微博和网易微博都在使用这个库）是如何实现的

```
// 生成长度为B bits的随机私钥，E就是RSA公钥(n,e)中的那个e，通常取3或者65537
function RSAGenerate(B,E) {
  var rng = new SecureRandom();
  //私钥的长度为B bits，相应的p和q的长度大概为B/2
  var qs = B>>1;
  this.e = parseInt(E,16);
  var ee = new BigInteger(E,16);
  for(;;) {
    //生成质数p
    for(;;) {
      this.p = new BigInteger(B-qs,1,rng);
      //这里利用了e与p-1互质的特性
      if(this.p.subtract(BigInteger.ONE).gcd(ee).compareTo(BigInteger.ONE) == 0 &&
this.p.isProbablePrime(10)) break;
    }
    //生成质数q
    for(;;) {
      this.q = new BigInteger(qs,1,rng);
      //这里利用了e与q-1互质的特性
      if(this.q.subtract(BigInteger.ONE).gcd(ee).compareTo(BigInteger.ONE) == 0 &&
this.q.isProbablePrime(10)) break;
    }
    if(this.p.compareTo(this.q) <= 0) {
      var t = this.p;
      this.p = this.q;
      this.q = t;
    }
    var p1 = this.p.subtract(BigInteger.ONE);
    var q1 = this.q.subtract(BigInteger.ONE);
    var phi = p1.multiply(q1);
    if(phi.gcd(ee).compareTo(BigInteger.ONE) == 0) {
      this.n = this.p.multiply(this.q);
      this.d = ee.modInverse(phi);
      this.dmp1 = this.d.mod(p1);
      this.dmq1 = this.d.mod(q1);
      this.coeff = this.q.modInverse(this.p);
      break;
    }
  }
```

明白了原理，是不是觉得非常简单呢？^_^

#0x03 RSA公司的节操是如何失掉的 小伙伴们一定还记得伟大的斯诺登同志(-_-)在危急关头挺身而出，曝光了传说级的“棱镜计划”。在这个事件的后续报道中，不知道大家有没有注意到这样的一则报道

> 据NSA前通讯员斯诺登所提供的机密文件显示，NSA跟RSA达成了一份价值1000万美元的合同，前者通过在后者的加密软件中植入一个缺陷公式，为自己留了一道“后门”。据悉，RSA存有缺陷公式的软件叫做Bsafe，而该缺陷公式的名字为Dual Elliptic Curve，它由NSA开发而出。文件内容指出，RSA自2004年起就开始在自己的软件中使用了这一缺陷公式。

那么这个有缺陷公式到底是怎么回事呢？其实也不是什么新鲜事物，而且与大素数的生成还有那么一点点联系~

> RSA Security是由RSA算法的发明者Ron Rivest, Adi Shamir和Len Adleman在1982年创立，随后在2006年以21亿美元的价格被EMC公司收购。其中该算法最为有名的一个缺陷就是DUAL_EC_DRBG，密码学家早在几年前就发现了这个问题。这个加密算法可以看作是随机数生成器，但是有些数字是固定的，密码学家能够将其作为万能钥匙通过一些内置的算法进行破解。

我们知道RSA算法本身没什么问题，因为只要你的密钥是真正随机产生的，猜对这个密钥就如同大海捞针一般难，现有计算机肯定无法在密码更换周期内攻破你的加密档案。但是，如果这个随机算法是假的呢？如果它仅仅是在一个很小的集合中产生的密钥呢？你的加密档案瞬间就被查看了，这就是NSA干的事情。

如果小伙伴们有兴趣的话可以参考比利时计算机科学专家Aris Adamantiadis发表的Dual_EC_DRBG后门的概念验证研究报告

0x04 总结

大素数生成过程有两个关键点，一个是素性测试，另一个是随机数的生成。前者要保证正确性和高效性，后者则一定要保证安全性，即生成一个真随机数，而并非坑爹的RSA干的事(-_-)

##其他素性测试 说了这么多，到底有没有一个确定的素性测试算法呢？（除了试除法-_-）当然有了~这就是传说中的AKS素性测试

> AKS素数测试（又被称为Agrawal–Kayal–Saxena素数测试和Cyclotomic AKS test）是一个决定型素数测试算法，由三个来自Indian Institute of Technology Kanpur的计算机科学家，Manindra Agrawal、Neeraj Kayal和Nitin Saxena，在2002年8月6日发表于一篇题为*PRIMES is in P*（素数属于P）的论文。作者们因此获得了许多奖项，包含了2006年的哥德尔奖和2006年的Fulkerson Prize。这个算法可以在多项式时间之内，决定一个给定整数是素数或者合数。

关于这个算法小伙伴们可以去看他们的论文，我只想说：论文的标题还能再牛X一点么？！

关于真随机

大部分程序和语言中的随机数（比如`C`和`MATLAB`中的），确实都只是伪随机。是由可确定的函数（比如线性同余），通过一个种子（比如时钟），产生的伪随机数。不过`UNIX`内核中的随机数发生器（`/dev/random`），它在理论上能产生真随机。即这个随机数的生成，独立于生成函数，或者说这个产生器是非确定的。所以，**计算机理论上可以产生统计意义上的真随机数**(-_-)

V2EX Contact me at: 知
Site powered by Jekyll & Github Pages. Theme designed by HyG.